

[**LINUX MAGAZINE**](#) • [subscribe](#) • [advertise](#) • [customer service](#) • [back issues](#) • [feedback](#) • [contacts](#)

[search](#)

Newsletter

Email:

Name:

[sign up](#)

J/XFS Devices Services

devices services for ongoing J/XFS kernel

Solaris Kernel Tuning

Automated recommendation explanations for Solaris, AIX

Ad

[Linux Magazine / July 1999 / FEATURES](#)
Secrets Inside the Linux Kernel Revealed

[<< prev](#) page 1 2 3 4 [next >>](#)

[Ads by Goooooogle](#)

Dsp Kernel

Operating Systems
Intel® Developer Services
www.intel.com

Understand Linux Kernel

Buy Daniel P. Bovet's book Now at 30% off.
associate
www.amazon.com

Linux

Rocks and Rolls!
linspire.com

4-way Itanium® 2 Systems

4U: 1.5GHz w.
6MB Cache, 32GB RAM Configure & Price online at ION
itanium.ioncomputer.com

Virtual Memory

Earlier, I said that the virtual memory subsystem in the kernel was responsible for memory protection (preventing processes from interfering with each other's memory) and paging (allowing memory pages to be temporarily moved to disk). This is easily one of the most complex aspects of the Linux kernel, and it bears taking a closer look at. Understanding how virtual memory works in Linux is the key to understanding many other features, such as the swap subsystem.

This is a topic which has consumed a fair number of pages in books on advanced operating systems textbooks (not to mention a slew of web pages and articles in magazines such as *Glamour*), so I can't expect to cover the bulk of virtual memory management in this article. Hopefully, this will give you a clear picture of what the various pieces are so you can look for more detail elsewhere. Different operating systems implement virtual memory management in very different ways, so my treatment here will be specific to Linux. It's also helpful to focus on a particular hardware architecture. Linux relies heavily (as do other operating systems) on the memory support provided by the hardware. In this case, I'll focus on the Intel x86 architecture, as the discussion is similar for other systems.

The very meaning of the term virtual memory is that an application sees the illusion that it has access to a much larger amount of memory than is present on the system. Ideally, we'd like the application to believe that it has access to the entire range of memory addresses allowed by the CPU -- which is 2^{32} bytes, or 4 gigabytes of virtual memory. (This is because every "virtual address", is stored in 32 bits and each bit only has 2 possible values.) The range of virtual memory which can be addressed by a pointer is called the "virtual address space". The virtual address space, then, is the range of virtual memory which can be addressed by a pointer. We can write this range, in hexadecimal, as 0x 00000000 to 0x FFFFFFFF. Very few systems, however, have 4 gigabytes of physical memory. In addition, multiple applications may be running on the system at the same time, so each application to be able to access the entire 4-gigabyte "virtual address space" without interfering with one another. How are we going to accom-

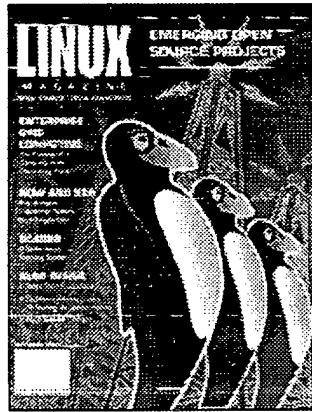
Luckily, the Intel x86 architecture (as well as most other modern architectures) includes hardware support for implementing virtual memory.

Sections

[Editorial](#)
[Newbies](#)

[Reviews](#)
[Shutdown](#)
[How To](#)
[LAMP Post](#)
[Power Tools](#)
[Guru Guidance](#)
[On The Desktop](#)
[Best Defense](#)
[Developer's Den](#)
[Java Matters](#)
[Extreme Linux](#)
[Compile Time](#)
[Perl of Wisdom](#)
[Gearheads Only](#)
[Project of the Month](#)
[Who's Who](#)
[Downloads](#)

On Stands Now



Indexes

[Issue Archive](#)
[Author Index](#)

RSS feed: [XML](#)

Subscribe Now!

Name:
 Address:
 Address2:
 City:
 State: XX (US only)
 Zip:
 Email:

12 issues for \$29.95
 24 issues for \$54.95

[Subscribe](#)

form of a "memory management unit", or MMU. The MMU is responsible for translating virtual memory accesses made by user programs into accesses of the actual RAM in the machine. Of course, the MMU is not able to do this on its own -- it needs the operating system to do this -- which is where the Linux kernel's memory management subsystem comes into play. Before we get into all of that, however, let's look at what happens when a user program reads or writes an address in memory.

Translating virtual addresses

Let's say that a user process (such as Emacs) wants to read or write to the memory at virtual address 0x 48b32f00 (this address is meaningless to a human, but to Emacs it might represent the current position of the cursor). In order to translate this memory reference into an address in physical memory, the MMU uses a special set of structures, called the page tables. Each page table entry specifies for each page (4 kilobytes) of virtual memory what physical address (if any) should be used. We can think of a particular page containing the following information:

virtual page address → physical page address (+ other information)

So, there might be an entry in the page tables which looks like:

0x48b32000 → 0x0028a000 (+ other information)

Note that the MMU deals with memory in page-sized chunks, which (for things) reduces the size of the page tables themselves. Since a page table entry is 4 bytes long, the last 3 digits (in hexadecimal) of the virtual and physical page addresses are always '000'. The last three digits of an address (such as 0x48b32000) are the offset into the page which the user is accessing; in this case that means that the user is accessing the 28a000th byte of the page. The MMU adds the page offset to the physical page address found in the page table to produce a complete physical memory address -- *voila!*

Certainly not every virtual address in the entire 4 GB range corresponds to a physical memory address -- unless one has 4 GB of memory installed. In fact, these page table entries might contain a physical page address which is marked as "invalid", meaning that there is no physical page corresponding to that address. When an invalid entry is read by the MMU, a page fault occurs. We'll talk about this case later.

Page tables and the TLB

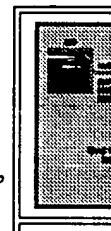
Note that the page tables are actually stored in memory themselves, which is an interesting issue, as it means that the MMU might have to consume a significant amount of memory in order to look up something else in the page tables. This issue is compounded by the fact that there is not just a single page table -- rather, there are thousands of page tables which must be traversed by the MMU in order to translate a virtual address into a physical address. This is done for space reasons; in fact, a single page table entry actually consumes four bytes. If we have thousands of page tables, we'll need a lot of memory to store them all.

Non-US orders? Click here for delivery to:
[Canada](#) • [Mexico](#) • [Other](#)

entry per page of virtual address space, that's $((4\text{ Gb}/4\text{ Kb}) * 4\text{ b}$ megabytes of memory, just to hold the page tables for a single user. Multiple levels of page tables actually allows the hardware to swap page tables out to disk when they're not being used -- we warned you going to get hairy! For now, though, don't worry about it -- it suffices to say that there is a single (in-memory) page table being consulted by every memory reference.

If the MMU is performing multiple memory operations just to translate an address for another memory operation, clearly this is going to be a performance hit. To remedy this problem, the MMU hardware includes a translation lookaside buffer (TLB) for virtual-to-physical memory translations, called the translation lookaside buffer or TLB. The TLB can be thought of as containing a very small page table in very fast RAM to speed up MMU address translations.

Figure 2 shows what happens during a single memory access by a user application. On the upper left of the figure we have the CPU, from which a user process wishes to read the virtual address 0x48b32f00. First, the TLB is consulted, and if a translation is found there, the physical address is used to access memory directly. (We've drawn the TLB as being separate from the CPU itself, but it's actually part of the processor in most cases.) If the TLB does not contain a mapping for the given virtual address, the MMU must then perform the arduous task of looking up the address in the page tables. However, the result of this lookup will be saved in the TLB, increasing performance if another address on the same page is accessed.



How the access from a address

So far I've talked about the Intel x86's MMU hardware, not about the Linux kernel. Clearly, the kernel isn't involved every time a virtual address is translated -- it would be far too slow. So, what does the kernel have to do with memory management?

The first job of the kernel is to set up and maintain the page tables that the MMU will traverse when translating addresses. This requires the kernel to know how physical memory is laid out, to allocate portions of it to various processes, and to create page table entries allowing virtual addresses to be mapped onto physical RAM.

[<< prev](#) [page 1 2 3 4](#) [next >>](#)

*Linux Magazine / July 1999 / FEATURES
Secrets Inside the Linux Kernel Revealed*

[LINUX MAGAZINE](#) • [subscribe](#) • [advertise](#) • [customer service](#) • [back issues](#) • [feedback](#) • [contacts](#)

This Page Blank (uspto)